

LARGE CODE BASE:

x This section merely provides guidelines, it is not written to follow any formal structure.

1. Read through:

- Documentation and examples, while using examples read API and browse through the source code, scanning the syntax, style of writing, deducing any patterns, etc;
- Read API thoroughly, imagine what the functionalities do; etc;
- Call the developers and ask questions about the system – design patterns, etc
- Determine if you are reading a framework or class of libraries. Patterns are still necessary to integrate with both, like one can use the strategy pattern on Django views to control the different model engines.

2. Coding principles:

- Is it functional or object oriented style;
- Verify that the principles have been followed: SOLID;
 - x Check for inheritance, abstarctions and dependencies – especially dependency injections; It very critical to fully understand the problem being modelled so that we can easily define the classes/objects with their attributes and methods. Look for over-engineered solutions, solutions should be simple because requirements should be stated clearly. It is of no need to waste resources.*
 - x **Read the codebase thoroughly, investigate it, follow it through manually, read documentations; then, use tools, Doxygen, to fill up the gaps.***
 - x Find out how you will get it work, when given a problem get it to work first; in these you'll get to establish the critical data and methods required.*
 - x Normally the concrete instances that varies are grouped together under an interface, abstract class or base class (inheritance), then a client class is created which combines the concrete objects in all sorts of different ways (normally via composition) and it defines its method interfaces' which calls the concrete objects' implementation method.*
 - x Whether abstract classes are being used to manage subclasses, and a separate interface – mostly abstract classes are used to provide homogenous objects with a common interface;*
 - x An interface must implement its own user intuitive methods, and through dependency injction calls the methods implemented by the subclasses that inherits the methods from the abstract class;*
 - x Class methods can be used to break down an __init__ with modalities and create each instance separately for the code to modular and improve unit testing;*
 - x Composition (initialise a class in an __init__ of another class) and Inheritance are the two standard techniques of reusing methods provided by the other classes;*
 - x One can also pass a class into a method for dependency injection techniques and the class will be local to the object;*
 - x Study the examples and tests. Inspect the classes or functions imported or locally defined that are involved in executing a particular example or test. That serves as an entry point to understand a particular functionality/component – you can use that to run gdb/pdb or to look up on Doxygen UML diagrams.*
 - x Through tests and examples you get to learn how to use the classes or framework. It is also important to establish why the classes/framework was written, for hugging*

face the objective is to create models, for keras to create models, for django to run a web framework, etc; then you can begin your tests in those modules.

- x UML Diagrams provide a structure, an arrangement of classes, from it we can then deduce the design approach for the problem.
 - x From all practical perspective software problems should be solved by iteration, different design, ad-hoc methods should be tried, tested and compared and select the one that closely satisfies the requirements.
 - x **Try understand each class libraries (subsystem) independently, focusing on the attributes, methods and local variables accepted. Then try to figure out the glue code between them to construct a framework, if it is a framework – how the classes use each other. Mostly this is found on examples and tests. Like how do we use the object provided? Glue code can be messy at first because it is meant to get the job done, and it mostly involves runtime objects – its getting the code to run! Normally this important glue code is hidden in utils, configs, decorators, etc – it is delegated some modules.**
 - x Use `grep -r "import module_name" path/to/source_code` to search where a particular module has been used in the source code.
- Check functions/methods:

Understand why a function is there and what it accepts as inputs and returns as an output; Check for data inputs and modality inputs;

A function operates on the stack and returns values, if not, it modifies some aspect or interacts with the external world;

Functions should be written to minimise call stack to avoid cascading dependencies – use `__main__` attribute to control the function at one place, or modalities with default values;

We can also opt to write a separate piece of function that evaluates the modalities (separate functions), and inject the function (call the function directly from the main function) - it is still the same idea inline with dependency injection (we inject the functionalities into a client/interface function).

The modalities functionalities are not performed inside the function itself – interface and implementation have been decoupled. Break down the function into units even for testing purposes.

Functions may include loop and conditionals statements as part of their logic, data checks and modality checks.

Is the codebase designed as a library of classes or a framework? A library of classes contains distinct classes performing various functionalities mostly unrelated, and frameworks provide classes that already have glue code to coordinate between them and the application developer merely writes code to “complete” the framework – he provides missing pieces for the river to flow, like Django.

3. Runtime environment:

- Try isolate the runtime environment to understand how the software works and which functions are invoked.

There would be so many paths involved – think about it like a Gant Chart – some will be critical, others not;

Since examples and tests actually implement/test a particular feature that serves as an entry point – trace back the function calls to the one that actually retruns the object;

There will be many hotspots in the code – find the functions that are called/call the most and try understand if they have a relationship;

4. Errors and constraints:

- From a library perspective, you want to ensure that you supply your users with useful error messages and to control how they use your objects.
You want to put fences around your objects and dictate how they should be used;
- From a user perspective you want to provide messages to those who will be running your scripts.

Tools:

1. Doxygen, ctags and cscope.
2. Sourcegraph and Github/1s/.
3. Sublime Text – LSP package with its language servers.

HOW GUNICORN WORKS - Version: 21.2.0

1. Fire up the terminal and type: `gunicorn <app> <host:port>`
2. Gunicorn has default settings for all the options, represented as classes in the file `config.py`. The list global variable: `KNOWN_SETTINGS` in the module `gunicorn.config` is populated via `gunicorn.config.SettingMeta metaclass`. Each class contains its default settings as class variables. *Metaclass* code snippet is given by:

A note on metaclasses:

- *Metaclasses are blueprints for creating classes and classes are blueprints for creating class instances/objects. They are commonly used for error handling and enforcing constraints on the user as to how the user should use the classes where a particular metaclass is defined. They are not executed during object instantiation, however during class definition, so the `KNOWN_SETTINGS` variable will be initialised during module import or this line: `Setting = SettingMeta('Setting', (Setting,), { })`.*

Figure Metaclass

```
2 def __new__(cls, name, bases, attrs):
3     super_new = super().__new__
4     parents = [b for b in bases if isinstance(b, SettingMeta)]
5     if not parents:
6         return super_new(cls, name, bases, attrs)
7
8     attrs["order"] = len(KNOWN_SETTINGS)
9     attrs["validator"] = staticmethod(attrs["validator"])
10
11     new_class = super_new(cls, name, bases, attrs)
12     new_class.fmt_desc(attrs.get("desc", ""))
13     KNOWN_SETTINGS.append(new_class)
14     return new_class
```

Options's class example code snippet is given by:

Figure Workers(Setting)

```
2 class Workers(Setting):
3     name = "workers"
4     section = "Worker Processes"
5     cli = ["-w", "--workers"]
6     meta = "INT"
7     validator = validate_pos_int
8     type = int
9     default = int(os.environ.get("WEB_CONCURRENCY", 1))
10    desc = """\
11        The number of worker processes for handling requests.
12
13        A positive integer generally in the "2-4 x $(NUM_CORES)" range.
14        You'll want to vary this a bit to find the best for your particular
15        application's work load.
16
17        By default, the value of the "WEB_CONCURRENCY" environment variable,
18        which is set by some Platform-as-a-Service providers such as Heroku. If
19        it is not defined, the default is "1".
20    """
```

The above class is responsible for the option which sets the number of workers in the command line, and it is set to: `default = int(os.environ.get("WEB_CONCURRENCY", 1))` as default (if the option isn't used)

3. The module `gunicorn. __main__.py` is executed and it imports the function `run` from `gunicorn.app.wsgiapp`.

The `run` function initialises the `gunicorn.app.wsgiapp.WSGIApplication("%(prog)s [OPTIONS] [APP_MODULE]")` object. This object inherits from `gunicorn.app.base.Application` which in turn inherits from `gunicorn.app.base.BaseApplication`, and the following is `BaseApplication's self.__init__` initialisation signature method :

Figure __init__ :

```
2 def __init__(self, usage=None, prog=None):
3     self.usage = usage
4     self.cfg = None
5     self.callable = None
6     self.prog = prog
7     self.logger = None
8     self.do_load_config()
9
```

The above method will be executed to initialise the `WSGIApplication` object.

During the above (*WSGIApplication*) object's initialisation, the method *self.do_load_config* will be executed which in turn executes:

Figure load_default_config:

```
2 def load_default_config(self):
3     # init configuration
4     self.cfg = Config(self.usage, prog=self.prog),
5
```

The above method initialises the *unicorn.config.Config* object to handle various configurations and set it to *self.cfg* attribute. The following is *Config*'s class initialisation signature method, *self.__init__*:

Figure self.__init__:

```
2 def __init__(self, usage=None, prog=None):
3     self.settings = make_settings()
4     self.usage = usage
5     self.prog = prog or os.path.basename(sys.argv[0])
6     self.env_orig = os.environ.copy()
```

We observe that as part of its initialisation process, it executes *make_settings* function from *unicorn.config* and the function has the following definition:

Figure make_settings:

```
2 def make_settings(ignore=None):
3     settings = {}
4     ignore = ignore or ()
5     for s in KNOWN_SETTINGS:
6         setting = s()
7         if setting.name in ignore:
8             continue
9         settings[setting.name] = setting.copy()
10    return settings
```

The function *make_settings* is responsible for initialising the *KNOWN_SETTINGS* list data type global variable by executing each class' options and populating the class name (key) and its initialised object (value) by line: *setting = s()* into a dictionary, *settings*.

After calling `load_config_default`, `self.__init__` (from `gunicorn.config.Config`) then proceeds to call, `load_config` which is not implemented in the `BaseApplication` class, hence `load_config` method from `gunicorn.app.base.Application` will be executed to carry out the initialisation further. After the initialisation of the `WSGIApplication` its `self.run` method will be called. The `self.run` method will execute the line: `self.load()` which will execute the `self.load` method from the `BaseApplication` and it is not implemented in the `BaseApplication`, but implemented in `WSGIApplication` where its signature is given as follows:

Figure load:

```
2 def load(self):
3     if self.cfg.paste is not None:
4         return self.load_pasteapp()
5     else:
6         return self.load_wsgiapp()
```

The above function loads the wsgiapp through the method `self.load_wsgiapp` with the following signature:

Figure load_wsgiapp:

```
2 def load_wsgiapp(self)
3     return util.import_app(self.app_uri)
```

Using the `app_uri` (**akin to a uniform resource identifier in web development**) argument, the method above will load the relevant app. The `self.run` method from `Application` class will ultimately execute: `super().run()` from the `BaseApplication` class, which will then initialise the `Arbiter` class object from `gunicorn.arbiter`, and the `self.run` method is given as follows:

Figure self.run:

```
2 def run(self):
3     try:
4         Arbiter(self).run()
5     except RuntimeError as e:
6         print("\nError: %s\n" % e, file=sys.stderr)
7         sys.stderr.flush()
8         sys.exit(1)
```

At this stage of the startup process, the *gunicorn.arbiter.Arbiter* class will be initialised and it's initialisation *self.__init__* signature is given as follows:

Figure self. __init__ :

```
2  def __init__(self, app):
3      os.environ["SERVER_SOFTWARE"] = SERVER_SOFTWARE
4      print("Find out which app")
5      print(app)
6      self._num_workers = None
7      self._last_logged_active_worker_count = None
8      self.log = None
9
10     self.setup(app)
11
12     self.pidfile = None
13     self.systemd = False
14     self.worker_age = 0
15     self.reexec_pid = 0
16     self.master_pid = 0
17     self.master_name = "Master"
18
19     cwd = util.getcwd()
20
21     args = sys.argv[:]
22     args.insert(0, sys.executable)
23
24     # init start context
25     self.START_CTX = {
26         "args": args,
27         "cwd": cwd,
28         sys.executable
29     }
```

The above *self.__init__* method will proceed to execute *self.setup(app)* method, as part of its initialisation, passing through the application object, which is the *WSGIApplication* class object. The line: *self.cfg = app.cfg* in the *self.setup(app)* method (defined in the *Arbiter* class) will define the attribute *self.cfg* and assign it the value *app.cfg* which is the *Config* class in *gunicorn.config*, and had already been defined by the following method from *BaseApplication*.

Figure load_default_config:

```
2  def load_default_config(self):
3      # init configuration
4      self.cfg = Config(self.usage, prog=self.prog)
```

The method `setup(app)` will proceed to define the attribute: `self.worker_class = self.cfg.worker_class` where `self.cfg.worker_class` is the `@property` method from `Config` class and it is given as follows:

Figure worker class:

```
2  @property
3  def worker_class(self):
4      uri = self.settings['worker_class'].get()
5
6      # are we using a threaded worker?
7      is_sync = uri.endswith('SyncWorker') or uri == 'sync'
8      if is_sync and self.threads > 1:
9          uri = "gunicorn.workers.gthread.ThreadWorker"
10
11     worker_class = util.load_class(uri)
12     print(worker_class)
13     if hasattr(worker_class, "setup"):
14         worker_class.setup()
15     return worker_class
```

The above method will be executed and more details will be provided at a later stage (read on). The method `setup(app)` will then proceed to load the application using the lines code:

Figure self.wsgi:

```
2  if self.cfg.preload_app:
3      self.app.wsgi()
```

The above method will load the application passed in the command line arguments, i.e., `django.wsgi:application` in case when one is running a django application.

5. After the above initialisation of `Arbiter()` object, the `self.run()` method (from `gunicorn.app.base.BaseApplication`) will proceed and then execute `Arbiter.run()`, which will then executes the `self.start` method and in turn executes the following line of code in its method body: `self.LISTENERS = sock.create_sockets(self.cfg, self.log, fds)` and this line is responsible for creating listeners on various sockets file descriptors and the method `self.start` will also be responsible for initialising the `init_signals`.

The method `self.run` will then executes `self.manage_workers` which will execute `self.spawn_worker` **[The line of code responsible:**


```

2 worker = self.worker_class(self.worker_age, self.pid, self.LISTENERS
3         self.app, self.timeout / 2.0,
4         self.cfg, self.log)

```

links us to the *Worker* classes in: *gunicorn.workers*]. The *self.spawn_worker* method uses *self.worker_class* (*defined above*) defined during initialisation and assigned to the *worker_class* @property (**which is a first class method/function**) method in order to load the *Worker* from *workers.sync.Sync* { (Gunicorn provides various *Worker* classes. Including *gthread*, *ggevent*, *geventlet* and the default, *SyncWorker*, can be changed, based on this *gunicorn.util* function's line of code located in its body: *worker_class = util.load_class(uri)*) (this is defined in part by the *uri*, **this locates the worker class, its a uniform resource identifier**) passed to the *worker_class* method of *Config*} and it will be initialised by *Worker.__init__* from *gunicorn.workers.base* with the signature given by:

Figure init :

```

2 def __init__(self, age, ppid, sockets, app, timeout, cfg, log):
3     """
4     This is called pre-fork so it shouldn't do anything to the
5     current process. If there's a need to make process wide
6     changes you'll want to do that in "self.init_process()".
7     """
8     self.age = age
9     self.pid = "[booting]"
10    self.ppid = ppid
11    self.sockets = sockets
12    self.app = app
13    self.timeout = timeout
14    self.cfg = cfg
15    self.booted = False
16    self.aborted = False
17    self.reloader = None
18
19    self.nr = 0
20
21    if cfg.max_requests > 0:
22        jitter = randint(0, cfg.max_requests_jitter)
23        self.max_requests = cfg.max_requests + jitter
24    else:
25        self.max_requests = sys.maxsize
26
27    self.alive = True
28    self.log = log
29    self.tmp = WorkerTmp(cfg)

```

self.spawn_worker will proceed to execute the following block of code in its body:

```

1 try:
2     util._setproctitle("worker [%s]" % self.proc_name)
3     self.log.info("Booting worker with pid: %s", worker.pid)
4     self.cfg.post_fork(self, worker)
5     worker.init_process()
6     sys.exit(0)

```

The above block of code will then execute `gunicorn.base.Worker.init_process` using the line: `worker.init_process()`, this line will execute `self.load_wsgi` in its method body, which then executes the line `self.wsgi = self.app.wsgi()`, and the `wsgi()` method will be determined by the `self.app`, which is equal to `arbiter.setup(app)` and loads the proper wsgi handler. Finally, the `init_process` will execute: `self.run()` block of code given as follows [

```

2 # Enter main run loop
3     self.booted = True
4     self.run() ]

```

to start the server loop. The `self.run` method is not implemented by the base `Worker` class but on concrete workers, hence `self.run` will be executed from the subclass `gunicorn.workers.sync.SyncWorker`. `self.run` will proceed to execute `self.run_for_one`, which in turn will execute `self.accept` to accept the connection **[for more refer to the sockets python standard library]**. `self.accept` method will call: `self.handle(listener, client, addr)` method which will in turn execute the line in its body: `parser = http.RequestParser` which is used to initialise the process of receiving and parsing the message.

```

1
2 The RequestParser
3 (class RequestParser(Parser):
4     msg_class = Request)
5

```

`http.RequestParser` inherits from `gunicorn.http.parser.Parser` and **[here for the first time we see link to the gunicorn.http API] [This API requires further investigation to disassemble exactly what the Parser does]**.

7. We proceed with the `self.handle` method, which will then execute `self.handle_request` and this will in turn execute this line of code in its body: `resp, environ = wsgi.create(req, client, addr, listener.getsockname(), self.cfg)`, `wsgi.create` is a function in `http.wsgi` and it contains the following line in its body: `resp = Response(req, sock, cfg)` to initialise a response object, **[here for the first time observe the relationship between the Response class and the Workers class]**, and the following line of code follows: `responder = self.wsgi(environ, resp.start_response)` which utilises `self.wsgi` attribute that has been assigned to our `wsgi app` to commence with the response.

The *Response* class from *gunicorn.http.wsgi* has a *self.start_response* method which is passed as an argument to the above *self.wsgi* (**its an object of the first class**) method which will be responsible for commencing with the response message.

[The *gunicorn.http.wsgi.Response* class requires further investigation.]

GUNICORN CONCLUSIONS

- x In this section we wish to discuss the classes involved during this process and the design patterns used to in the Gunicorn framework.
- x We have to understand the strategy used by Gunicorn to parse command line parameters, handle environment variables and configuration files – how did Gunicorn implement these steps.
- x How socket listening works with multiple threads and differing server architectures from standard ones, like gthread to third-party ones like gevent.
- x How it parses the request – Parser class, how does this class work.
- x And how it prepares responses – Response class, how does this class work.

HOW DJANGO WORKS BEHIND GUNICORN & DEVSERVER

- x We will exclude the database logic in the meantime: we will focus on: commands, servers, routing, views, forms, templates, middleware and request & response.
- x Insert `import pdb` and `pdb.set_trace()` inside functions, not outside them.

The Process begins

- ➔ We execute the *manage.py* file in the Django's project directory using the command line:
python manage.py runserver

Explanation:

- ➔ The function *manage.main()* is executed which then executes the function *django.manage.execute_from_command_line(sys.argv)*

Explanation:

Output:

- ➔ The following code snippet is executed from *django.core.management.__init__*

Figure execute from command:

```

1
2 def execute_from_command_line(argv=None):
3     """Run a ManagementUtility."""
4     utility = ManagementUtility(argv)
5     utility.execute()
6

```

Explanation:

This code snippet initialises the `django.core.management.__init__.ManagementUtility` class and then executes its method `execute()`. This `execute()` method runs some setups, which include creating the command line arguments parser and parsing the flags and arguments. Ultimately for the case of the `subcommand == runserver`, the following is executed: `self.fetch_command(subcommand).run_from_argv(self.argv)`

Output:

- ➔ The above (`execute_from_command_line`) will then execute the method `fetch_command(self, subcommand)`

Explanation:

This method is responsible for getting the application name (`app_name`) and the relevant subcommand which is (`runserver in this case`). It will then proceed to execute: `klass = load_command_class(app_name, subcommand)`, which is responsible for loading the relevant `Command` line class which handles `runserver`, the class is: `django.core.management.commands.runserver.Command(BaseCommand)`. This class handles the `runserver` subcommand, and it inherits from `BaseCommand`. Time to shed some explanation as of the design of Django's command line execution: We have the `BaseCommand` class with which all the other subcommands like, `runserver`, `makemigrations`, `migrate`, etc, inherits from. This design pattern encourages *flexibility and extensibility* because should we it allows to add more subcommands with minimal system disturbance, we just write concrete subclasses for the relevant option.

- ➔ The above `fetch_command()` method was called by `execute()` and it returns the above initialised `klass` for the `runserver` subcommand, and `execute()` proceeds to call: `run_from_argv(self.argv)` which is a `klass` method.

Explanation:

This function is not defined in the `runserver` instantiation of its `Command` class, `klass`, but it is inherited from `django.core.management.base.BaseCommand` class. `run_from_argv` will set up the environment (**remember for processes the information that we mostly manipulate is populated by the fork system call when a new process is being created, this indormation includes among: enviroment variables, command line arguments, and process's data structure information**), so during setup this is the information which is being manipulated.

- The above `run_from_argv` method will call the method: `self.execute(*args, **cmd_options)`

Explanation:

`execute()` is defined in the `BaseCommand` class and it will try to execute the line in its body: `subcommand == runserver`, by parsing some command line options first and even checking for migrations, it will then proceed to call the method: `output = self.handle(*args, **options)`

- The above `handle` method is not defined in the `BaseCommand` class:

Figure handle:

```
1
2 def handle(self, *args, **options):
3     """
4     The actual logic of the command. Subclasses must implement
5     this method.
6     """
7     raise NotImplementedError(
8         "subclasses of BaseCommand must provide a handle() method"
9     )
10
```

Which makes sense since the `BaseCommand` class is not a concrete class to execute subcommand's commands, so by the *object oriented programming* MRO algorithm the `handle` method from `runserver` concrete class will be executed.

- The above `self.handle` method will proceed with some initialisations and ultimately calls `self.run()` method.
- `self.run()` method will execute `self.inner_run()` both from `runserver's Command`. `self.inner_run()` will be responsible for getting the handler and executing run.

Figure run:

```
1
2 def run(self, **options):
3     """Run the server, using the autoreloader if needed."""
4     use_reloader = options["use_reloader"]
5
6     if use_reloader:
7         autoreload.run_with_reloader(self.inner_run, **options)
8     else:
9         self.inner_run(None, **options)
10
```

Explanation:

*handler = self.get_handler(*args, **options)*, this will execute *get_internal_wsgi_application()* from *django.core.servers.basehttp*, which will then execute:

Figure get_wsgi_application:

```
1
2 def get_wsgi_application():
3     """
4     The public interface to Django's WSGI support. Return a WSGI callable.
5
6     Avoids making django.core.handlers.WSGIHandler a public API, in case the
7     internal WSGI implementation changes or moves in the future.
8     """
9     django.setup(set_prefix=False)
10    return WSGIHandler()
11
```

To retrieve the *WSGIHandler* instantiated object from *django.core.handlers.wsgi*, and its signature is given as follows:

Figure self. __init__ :

```
1
2 def __init__(self, *args, **kwargs):
3     super().__init__(*args, **kwargs)
4     self.load_middleware()
5
```

This class (*WSGIHandler*) is responsible for loading middleware objects from the class *django.core.handlers.base.BaseHandler* via the method *self.load_middleware()*.

[Middleware is a vast section which we will return to for a thorough discussion. For now let's proceed with starting our django development server].

- ➔ Ultimately the initialised *WSGIHandler* object will be returned to *self.inner_run*, which will then proceed to execute the *run* function from *django.core.servers.basehttp*, and this will be responsible for starting our development server.
- ➔ The *run()* function:

Explanation:

Its signature is given by:

Figure run:

```
1
2 run(
3     addr,
4     port,
5     wsgi_handler,
6     ipv6=False,
7     threading=False,
8     on_bind=None,
9     server_cls=WSGIServer,
10 ):
11
```

wsgi_handler argument is the *WSGIHandler* object obtained above, and our *server_cls* is the: *server_cls = WSGIServer*, this variables are *class variables* defined in *Command(BaseCommand)* class for *runserver* including *port*, *addr*, etc. This *run()* function will proceed to define: *httpd_cls = server_cls*, an *httpd_cls* local variable and initialises it to the *WSGIServer*. It will then define an *http* variable and initialises it to: *httpd_cls(server_address, WSGIRequestHandler, ipv6=ipv6)*, since *http_cls* is actually a *WSGIServer* object (**first class object**) to be executed and its initialisation signature is:

Figure self. __init__ :

```
1
2 def __init__(self, *args, ipv6=False, allow_reuse_address=True, **kwargs):
3     if ipv6:
4         self.address_family = socket.AF_INET6
5     self.allow_reuse_address = allow_reuse_address
6     super().__init__(*args, **kwargs)
7
```

The *WSGIServer* class defined in *django.core.servers.basehttp* inherits from the Python standard library *simple_server.WSGIServer* and this class's signature will also be executed by the above *super()* function. The *WSGIServer* from *simple_server* in turn inherits from *http.server.HTTPServer* from the Python standard library which then inherits from *socket_server.TCPServer* also from the Python standard library, which in turn inherits from *socket_server.BaseServer*, and ultimately this is the signature that will be executed:

Figure self. __init__ :

```
1
2 def __init__(self, server_address, RequestHandlerClass):
3     """Constructor. May be extended, do not override."""
4     self.server_address = server_address
5     self.RequestHandlerClass = RequestHandlerClass
6     self.__is_shut_down = threading.Event()
7     self.__shutdown_request = False
8
```

Notably, the *RequestHandlerClass* argument, in our case this class was bundled into the **args* argument which is a tuple *args* and unpacked again using **args* passed to *super().__init__(*args, **kwargs)* to positional arguments. In this case our *RequestHandlerClass* argument is: *WSGIRequestHandler* and this class defined in *django.core.servers.basehttp*. In this whole process our request handler will be initialised into: *self.RequestHandlerClass = RequestHandlerClass(WSGIRequestHandler)*

- ➔ Proceeding with the *run()* function it will also call: *httpd.set_app(wsgi_handler)* to setup the app. This function will execute the method in:

Figure set_app:

```
1
2 def set_app(self,application):
3     self.application = application
4
```

This will setup our application to *wsgi_handler*, which is our *WSGIHandler* class from *django.core.handlers.wsgi*.

- ➔ Then *run()* will execute: *httpd.serve_forever()*, this will execute *self.serve_forever()* from *socket_server.BaseServer*, the Python standard library:

Explanation:

self.serve_forever will use the *selectors* standard library module to register sockets of interest with their corresponding events, [refer to **selectors module from the Python standard library for more in-depth explanation of how it operates**].

And it will then call: *self._handle_request_noblock()*

- ➔ *self._handle_request_noblock()* will call: *request, client_address = self.get_request()*, which will then call *accept* from Python sockets standard library to accept a connection, and this function is in the subclass *TCPServer*. After the connection

- ➔ has been accepted, the method will then call: `self.process_request(request, client_address)`, which will call: `self.finish_request(request, client_address)`.
- ➔ The method `self.finish_request` will initialise our *RequestHandler* class using the line in its body: `self.RequestHandlerClass(request, client_address, self)`

Explanation:

Initialisation of the *RequestHandlerClass* which is the *WSGIRequestHandler* from `django.core.servers.basehttp`, this class inherits from `wsgiref.simple_server.WSGIRequestHandler(BaseHTTPRequestHandler)`, from the Python standard library which then inherits from *BaseHTTPRequestHandler* from the same module, which then inherits from `socket_server.StreamRequestHandler(BaseRequestHandler)`, from the Python standard library which then inherits from *BaseRequestHandler* from the same module, and this class has an initialisation signature:

Figure self. __init__ :

```
1
2 def __init__(self, request, client_address, server):
3     self.request = request
4     self.client_address = client_address
5     self.server = server
6     self.setup()
7     try:
8         self.handle()
9     finally:
10        self.finish()
11
```

Ultimately the above is what will be executed. The *self.handle* method will be executed from `django.core.servers.basehttp.WSGIRequestHandler` *self.handle()* method.

- ➔ The above `self.handle()` method will call: `self.handle_one_request()`
- ➔ And `self.handle_one_request()` will:

Explanation:

This method will ultimately execute the following important lines of code in its body:

```

1
2 handler = ServerHandler(
3     self.rfile, self.wfile, self.get_stderr(), self.get_environ()
4 )
5 handler.request_handler = self # backpointer for logging & connection
closing
6 handler.run(self.server.get_app())
7

```

ServerHandler will be initialised and this class is in *django.core.servers.basehttp* and it inherits from *wsgiref.simple_server.ServerHandler* which then inherits from *wsgiref.handlers.SimpleHandler*, both from the Python standard library, with the following initialisation signature:

Figure self. __init__ :

```

1
2 def __init__(self, stdin, stdout, stderr, environ,
3     multithread=True, multiprocess=False
4 ):
5     self.stdin = stdin
6     self.stdout = stdout
7     self.stderr = stderr
8     self.base_env = environ
9     self.wsgi_multithread = multithread
10    self.wsgi_multiprocess = multiprocess
11

```

and it inherits from *BaseHandler* from the same module (The Python standard library). This line: *handler.run(self.server.get_app())* will get the application which is a *WSGIHandler* application from *django.core.handlers.wsgi* module. And it will the execute *self.run* method from *BaseHandler*:

Further details

→ **Figure run:**

```
1
2 def run(self, application):
3     """Invoke the application"""
4     # Note to self: don't move the close()! Asynchronous servers shouldn't
5     # call close() from finish_response(), so if you close() anywhere but
6     # the double-error branch here, you'll break asynchronous servers by
7     # prematurely closing. Async servers must return from 'run()' without
8     # closing if there might still be output to iterate over.
9     try:
10         self.setup_environ()
11         self.result = application(self.environ, self.start_response)
12         self.finish_response()
13     except (ConnectionAbortedError, BrokenPipeError, ConnectionResetError):
14         # We expect the client to close the connection abruptly from time
15         # to time.
16         return
17     except:
18         try:
19             self.handle_error()
20         except:
21             # If we get an error handling an error, just give up already!
22             self.close()
23             raise # ...and let the actual server figure it out.
24
```

This method has been reproduced here because it is the method that links the *Django application and the server*. It will setup the environment and calls: *self.result = application(self.environ, self.start_response)*, where *application* is our *WSGIHandler*. This call will execute *__call__(self, environ, start_response)* and its signature is given by:

Further details

Figure self. call :

```
1
2 def __call__(self, environ, start_response):
3     print("Who handles the response")
4     print(start_response)
5     set_script_prefix(get_script_name(environ))
6     signals.request_started.send(sender=self.__class__, environ=environ)
7     request = self.request_class(environ)
8     response = self.get_response(request)
9
10    response._handler_class = self.__class__
11
12    status = "%d %s" % (response.status_code, response.reason_phrase)
13    response_headers = [
14        *response.items(),
15        *((("Set-Cookie", c.output(header=""))) for c in response.cookies.values()),
16    ]
17    start_response(status, response_headers)
18    if getattr(response, "file_to_stream", None) is not None and environ.get(
19        "wsgi.file_wrapper"
20    ):
21        # If 'wsgi.file_wrapper' is used the WSGI server does not call
22        # .close on the response, but on the file wrapper. Patch it to use
23        # response.close instead which takes care of closing all files.
24        response.file_to_stream.close = response.close
25        response = environ["wsgi.file_wrapper"](
26            response.file_to_stream, response.block_size
27        )
28
```

return response from *WSGIHandler* thereby commencing with our response. The argument of interest into this function is *start_response* which is a function itself located in: *wsgiref.handlers.BaseHandler* class, and it is called by *__call__* above: at this line: *start_response(status, response_headers)*.

➔ This in a nutshell is the process that happens from the command: **python manage.py runserver** until we start servicing the request. In the next section we begin exploring servicing the request.

➔ Conclusions:
Design patterns:

Loose coupling/Dependency Injection is one of the fundamental design principles of object-oriented programming and it is followed in *Django*. Abstract classes are defined and concrete classes inherit from them and define their specific behaviour. APIs are provided which act like pivots to call all other functionalities and instantiate other objects. The method *self.__call__* executed above acts like a pivot (central API), where request & response functionalities are called from.

Servicing the request

- ➔ In the previous section we have entered the `self.__call__` method of `WSGIHandler` class from `django.core.handlers.wsgi`. This class has a class variable: `request_class = WSGIRequest`, which is then called on this line in method's body: `request = self.request_class(enviro)`. This will begin the preparation of an `HttpRequest` (the famous request we pass into our views) object.
- ➔ The `WSGIRequest` class inherits from `HttpRequest` class in `django.http.request`. This class will be initialised (**its initialisation is a process and we will get back to it**), and the initialised request object will be stored in `request` local variable.
- ➔ Then a call is made to: `response = self.get_response(request)` to `self.get_response` method with `request` object as an argument. Since `WSGIHandler` inherits from `django.core.handlers.base.BaseHandler` this method will be executed from this class, and we reproduce it here:

Figure get_response:

```
1
2 def get_response(self, request):
3     """Return an HttpResponse object for the given HttpRequest."""
4     # Setup default url resolver for this thread
5     set_urlconf(settings.ROOT_URLCONF)
6     response = self._middleware_chain(request)
7     response._resource_closers.append(request.close)
8     if response.status_code >= 400:
9         log_response(
10             "%s: %s",
11             response.reason_phrase,
12             request.path,
13             response=response,
14             request=request,
15         )
16     return response
```

- ➔ The `self.get_response` method will execute, `response=self._middleware_chain(request)` where `self._middleware_chain` is a list of `middleware` objects initialised by the method `self.load_middleware` of the `BaseHandler` class. This list contains various middleware classes which must be executed to inspect the request object, they include among: `django.middleware.security` [**django.middleware.security (investigate more how the below method is called from the above)**].
- ➔ Ultimately, the method: `self._get_response(self, request)` and we reproduce part of it here since it plays a crucial role:

Figure get response:

```
1
2 def _get_response(self, request):
3     """
4     Resolve and call the view, then apply view, exception, and
5     template_response middleware. This method is everything that
6     happens
7     inside the request/response middleware.
8     """
9     response = None
10    callback, callback_args, callback_kwargs =
self.resolve_request(request)
11    # Apply view middleware
12    for middleware_method in self._view_middleware
13        response = middleware_method(
14            request, callback, callback_args, callback_kwargs
15        )
16    if response:
17        break
18
19    if response is None:
20        wrapped_callback = self.make_view_atomic(callback)
21        # If it is an asynchronous view, run it in a subthread.
22        if iscoroutinefunction(wrapped_callback):
23            wrapped_callback = async_to_sync(wrapped_callback)
24        try:
25            response = wrapped_callback(request, *callback_args,
**callback_kwargs) # Where view is RUN!
26            print("final response")
27        except Exception as e:
28            response = self.process_exception_by_middleware(e, request)
29            if response is None:
30                raise
31
32    # Complain if the view returned None (a common error).
33    self.check_response(response, callback)
34
35    # If the response supports deferred rendering, apply template
36    # response middleware and then render the response
37    if hasattr(response, "render") and callable(response.render):
38        for middleware_method in self._template_response_middleware:
39            response = middleware_method(request, response)
40        # Complain if the template response middleware returned
None
41        # (a common error).
```

is called which is internally used by `self.get_response` to get the job done. This method will call: `callback, callback_args, callback_kwargs = self.resolve_request(request)` in

its body to resolve the request and returns among others in a tuple the *callback view function/class* defined in the application's *view.py* module. This method also has an asynchronous alternative depending on the *self.adapt_method_mode* method of *BaseHandler*.

→ The above mentioned method *self.resolve_request* calls *get_resolver* given as follows:

Figure get_resolver:

```
1
2 def get_resolver(urlconf=None) from django.urls:
3     if urlconf is None:
4         urlconf = settings.ROOT_URLCONF
5     return _get_cached_resolver(urlconf)
6
```

This function will call the function *_get_cached_resolver* from *django.urls.resolvers* which will return the initialised object of the class: *URLResolver(RegexPattern(r"^/"), urlconf)* of *django.urls.resolvers* (**This class must be discussed separately**).

Ultimately, the method *self.resolve* from the above *URLResolver* object will be called and it will return the initialised object:

```
1
2 ResolverMatch(
3     sub_match.func,
4     sub_match.args,
5     sub_match.dict,
6     sub_match.url_name,
7     [self.app_name] + sub_match.app_names,
8     [self.namespace] + sub_match.namespaces,
9     self._join_route(current_route, sub_match.route),
10    tried,
11    captured_kwargs=sub_match.captured_kwargs,
12    extra_kwargs={
13        **self.default_kwargs,
14        **sub_match.extra_kwargs,
15    },
16 )
17
```

of *django.urls.resolvers* [**This object warrants a discussion of its own**], and the following line from *self.resolve_request* will be executed: *request.resolver_match = resolver_match* which will assign *resolver_match* class variable to the above *resolver_match* object. And *resolver_match* tuple will be returned to *self._get_response* method of the *BaseHandler* class in *django.core.handlers.base*, the tuple is possible because of the *__getitem__* method in the *django.urls.ResolverMatch* class.

→ Now that we have our view function (**or class which needs further investigation**) which we can use to further the *self._view_middleware* checks, in this list we have middleware

objects like: *CsrfViewMiddleware*, *SessionMiddleware* and *SecurityMiddleware*, this middlewares include checking of the view itself and redirecting to the corresponding templates for quick responses i case certain checks fail. **[This middleware requires further investigation]**.

- ➔ This section is responsible in the method *self._get_response* is responsible for executing the view's checks using below code block:

```
1
2 if response is None:
3     wrapped_callback = self.make_view_atomic(callback)
4     # If it is an asynchronous view, run it in a subthread.
5     if iscoroutinefunction(wrapped_callback):
6         wrapped_callback = async_to_sync(wrapped_callback)
7     try:
8         response = wrapped_callback(request, *callback_args,
9 **callback_kwargs)
```

If response is *None* it confirms that middleware passed, otherwise a built-in template will have been populated and returned as a response to the request. This code block then calls: *make_view_atomic(self, view)* in the same class *BaseHandler*, then the line: *response = wrapped_callback(request, *callback_args, **callback_kwargs)* for synchronous execution is called to *execute the view*. Now we are in the *views.py* file in the application directory.

- ➔ Conclusion:

Design patterns:

Routing needs thorough investigations, and most of the concepts are missing here, especially the link between *get_response* and its executor *_get_response*.

Lets have a discussion about this method: *self.resolve* from *django.urls.resolvers.URLResolver*, This method is the pivot (the API) for URL resolving. It calls all other functionalities to return required objects, like:

```
@cached_property
def url_patterns(self):
```

and: *django.urls.resolvers.RegexPattern(CheckURLMixin)*. And the method *self.resolve* from *django.urls.resolvers.URLPattern*, where *URLPattern* is returned by the partial function: *path = partial(_path, Pattern=RoutePattern)* from *django.urls.conf*, which depend on the function: *_path(route, view, kwargs=None, name=None, Pattern=None)* also from *django.urls.conf* - this is how the function play a role when defining *urlpatterns* list in *urls.py* modules and also on the setting: *ROOT_URLCONF = "django_ref.urls"* from *settings.py* module.

Execution of the View

- ➔ From this line: `response = wrapped_callback(request, *callback_args, **callback_kwargs)`, if the view function has a `render` shortcut method, render will be executed from `django.shortcuts.render`.
- ➔ This is the `render()` function defined in `django.shortcuts`:

Figure render:

```
1
2 def render(
3     request, template_name, context=None, content_type=None, status=None,
4     using=None
5 ):
6     """
7     Return an HttpResponse whose content is filled with the result of calling
8     django.template.loader.render_to_string() with the passed arguments.
9     """
10    content = loader.render_to_string(template_name, context, request,
11                                     using=using)
12    return HttpResponse(content, content_type, status)
```

It begins by calling `loader.render_to_string` from `django.template.loader`, which calls: `template = get_template(template_name, using=using)` or calls `set_template` depending on some checks, both which are located in `django.template.loader`.

- ➔ `get_template` function calls `engines = _engine_list(using)`, `_engine_list` function which returns the engines being used: we have `DjangoTemplates` and `Jinja2` engines in the files `django.templates.backends.django` and `django.templates.backends.jinja2` respectively, which both inherit from `BaseEngine` from `django.templates.backends.base`. Then for each engine we will call its `self.get_template` method and it will return: `Template(self.engine.get_template(template_name), self)` in case of a django template. The `Template` class is specifically for the `Django` engine and it is located in: `django.templates.backends.django`.

Details must be observed in the initialisation of this class `Template`: the argument: `self.engine.get_template(template_name)` is of particular interest. This argument is actually the `Template` base class defined in: `django.templates.base` and it is assigned to `self.template = template` class variable of the `Template` class in `django.templates.backends.django`, it holds it as an extended class (**Composition principle of object oriented programming**). This class `Template` initialised to `self.template` it is responsible for parsing the template, we note that the module `django.templates.base` is 1122 lines long since it defines `Nodes` and `Parser` classes. [This needs a through investigation on how the django template parser works]

- ➔ After the *Template* object from *django.templates.backends.django* has been initialised to the class engine *DjangoTemplates* *get_template* method. Which is ultimately returned to the function *get_template* from *django.template.loader*, and which returns to the function *render_to_string* in *django.template.loader*, then the function will execute: *template.render(context, request)* the *render* method from *Template*.
- ➔ Here is the *self.render* method:

Figure self.render:

```

1
2 def render(self, context=None, request=None):
3     context = make_context(
4         context, request, autoescape=self.backend.engine.autoescape
5     )
6     try:
7         return self.template.render(context)
8     except TemplateDoesNotExist as exc:
9         reraise(exc, self.backend)

```

This method executes *make_context* from *django.template.context*, and this function is responsible for gathering and executing the context/dictionary passed into the template inside the view function/class [**This is requires further investigation as this is where forms come into play**]. Normally forms will be in *forms.py* module of the application as classes and initialised inside a view – its a vast topic found in *django.forms API*. However the function *make_context* interacts with the *Context* and *RequestContext* classes defined in: *django.template.context*.

- ➔ Then the *render()* function will call: *return self.template.render(context)* this is called on the *Template* from *django.templates.base* to render the template and parse it [**This is another world which requires further investigations**]. Ultimately the *render()* function from *django.shortcuts* will call:
- ➔ The class object *HttpResponse(content, content_type, status)* will be initialised from *django.http.response* and it will be called by the *render()* function from *django.shortcuts* to commence with the response, and after initialisation and having been returned to the *render()* function, it *render()* function will ultimately return to *self.get_response* from *django.core.handlers.base* together with the initialised *HttpResponse* object, which will then return to the method *self.__call__* from *django.core.handlers.wsgi* and stored in the local variable *response*.
- ➔ The method *self.__call__* will proceed to execute the line: *start_response(status, response_headers)*, which will execute the below method passed in as an argument to *self.__call__*:

Figure start_response

```
1
2 def start_response(self, status, headers, exc_info=None):
3     """start_response() callable as specified by PEP 3333"""
4     if exc_info:
5         try:
6             if self.headers_sent:
7                 raise
8         finally:
9             exc_info = None # avoid dangling circular ref
10    elif self.headers is not None:
11        raise AssertionError("Headers already set!")
12    self.status = status
13    self.headers = self.headers_class(headers)
14    status = self._convert_string_type(status, "Status")
15    self._validate_status(status)
16    if __debug__:
17        for name, val in headers:
18            name = self._convert_string_type(name, "Header name")
19            val = self._convert_string_type(val, "Header value")
20            assert not is_hop_by_hop(name), \
21                f"Hop-by-hop header, '{name}': {val}, not allowed"
22    return self.write
23
```

from `wsgiref.handlers.BaseHandler`. This method will return to the `self.run` method (From **Figure run** above) method from `wsgiref.handlers.BaseHandler` (as to who return sto who needs assessemnet)

(The above `self.write` seems to be a bit challenging to locate as to where it returns to `self.run` in `wsgiref.handlers.BaseHandle` or `__call__`? These details still need attention)

- ➔ And ultimately `self.__call__` will return the **response** object which will be assigned to `self.result` attribute of `wsgiref.handlers.BaseHandler` in the Python standard library to finish up the response and the method `self.run` (from **Figure run** above) will proceed to execute the line: `self.finish_response`, and according to the *MRO* algorithm, hence the `self.finish_response` that will execute is the one defined in: `django.core.servers.basethttp.ServerHandler` class.
- ➔ And the following is the code for the method `self.finish_response` from `django.core.servers.basethttp.ServerHandler`:

Figure finish_response:

```
1
2 def finish_response(self):
3     if self.environ["REQUEST_METHOD"] == "HEAD":
4         try:
5             deque(self.result, maxlen=0) # Consume iterator.
6             # Don't call self.finish_content() as, if the headers have not
7             # been sent and Content-Length isn't set, it'll default to "0"
8             # which will prevent omission of the Content-Length header with
9             # HEAD requests as permitted by RFC 9110 Section 9.3.2.
10            # Instead, send the headers, if not sent yet.
11            if not self.headers_sent:
12                self.send_headers()
13            finally:
14                self.close()
15        else:
16            super().finish_response()
17
```

The above will execute *self.close()* defined as:

Figure close:

```
1
2 def close(self):
3     self.get_stdin().read()
4     super().close()
5
```

- ➔ And the method *self.finish_response* it will ultimately call *super().finish_response* which is located in: *wsgiref.handlers.BaseHandler* and its definition is given by:

Figure finish response from `wsgiref.handlers.BaseHandler`:

```
1
2 def finish_response(self):
3     """Send any iterable data, then close self and the iterable
4
5     Subclasses intended for use in asynchronous servers will
6     want to redefine this method, such that it sets up callbacks
7     in the event loop to iterate over the data, and to call
8     'self.close()' once the response is finished.
9     """
10    try:
11        if not self.result_is_file() or not self.sendfile():
12            for data in self.result:
13                self.write(data)
14            self.finish_content()
15    except:
16        # Call close() on the iterable returned by the WSGI application
17        # in case of an exception.
18        if hasattr(self.result, 'close'):
19            self.result.close()
20        raise
21    else:
22        # We only call close() when no exception is raised, because it
23        # will set status, result, headers, and environ fields to None.
24        # See bpo-29183 for more details.
25        self.close()
```

- ➔ which will call close from `django.core.handlers.ServerHandler` and it is given by **(Review this might need to be removed, since there's a close already called from above and the other close below):**

Figure self.close:

```
1 def close(self):
2     self.get_stdin().read()
3     super().close()
```

- ➔ And this will ultimately call the close method in `wsgiref.handlers.BaseHandler` given by:

Figure close:

```
1 def close(self):
2     """Close the iterable (if needed) and reset all instance vars
3     Subclasses may want to also drop the client connection.
4     """
5
6     try:
7         if hasattr(self.result, 'close'):
8             self.result.close()
9     finally:
10        self.result = self.headers = self.status = self.envIRON = None
11        self.bytes_sent = 0; self.headers_sent = False
```

➔ Conclusions:

Design patterns:

[This section still requires attention]